

# Bounds for Compression in Streaming Models <sup>\*</sup>

**Abstract.** Compression algorithms and streaming algorithms are both powerful tools for dealing with massive data sets, but many of the best compression algorithms — e.g., those based on the Burrows-Wheeler Transform — at first seem incompatible with streaming. In this paper we consider several popular streaming models and ask in which, if any, we can compress as well as we can with the BWT. We first prove a nearly tight tradeoff between memory and redundancy for the Standard, Multipass and W-Streams models, demonstrating a bound that is achievable with the BWT but unachievable in those models. We then show we can compute the related Schindler Transform in the StreamSort model and the BWT in the Read-Write model and, thus, achieve that bound.

## 1 Introduction

The increasing size of data sets over the past decade has inspired work on both data compression and streaming algorithms. In compression research, the advent of the Burrows-Wheeler Transform [5] (BWT) has led to great improvements in both theory and practice. Streaming algorithms, meanwhile, are now used not only to process data online but also, because sequential access is so much faster than random access, to process them on disk more quickly. To combine these advances, it seems we must find a way to compute something like the BWT in a streaming model, e.g.:

1. **Standard:** In the simplest and most restrictive model, we are allowed only one pass over the input and memory sublinear (usually polylogarithmic) in the input's size (see, e.g., [23]).
2. **Multipass:** In one of the earliest papers on what is now called streaming, Munro and Paterson [22] proposed a model in which the input is stored on a one-way, read-only tape — representing external memory — that is completely rewound whenever we reach the end.
3. **W-Streams:** Ruhl [25] proposed a model in which we can also write: during each pass over the tape, we can replace its contents with something up to a constant factor larger.
4. **StreamSort:** Since we cannot sort in the W-Streams model with polylogarithmic memory and passes, Ruhl also proposed a generalization in which we can sort the contents of the tape at the cost of a constant number of passes (see also [1]).
5. **Read-Write:** Grohe and Schweikardt [13] noted that, with an additional tape, we can sort even in the W-Streams model; they proposed a model in which we have a read-write input tape, some number of read-write work tapes, and a write-only output tape.

---

<sup>\*</sup> Partially supported by the Italian MIUR and Italian-Israeli FIRB project “Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics”.

In a previous paper [12] we proved nearly tight bounds on how well we can compress in the Standard model with constant memory. We also showed how LZ77 [28] can be implemented with a growing sliding window to use slightly sublinear memory without increasing the bound on its redundancy, but that this cannot be done for sublogarithmic memory. Those bounds are, however, tangential to the common assumption of polylogarithmic memory. In this paper we consider whether the following bound can be achieved in the models described above with polylogarithmic resources: Given a string  $s$  of length  $n$  over an alphabet of constant size  $\sigma$ , can we store  $s$  in  $O(nH_k(s) + \sigma^k \log n)$  bits for all  $k$  simultaneously, where  $H_k(s)$  is the  $k$ th-order empirical entropy of  $s$ ? We first prove the bound unachievable in the first three models, via a nearly tight tradeoff between memory and redundancy. We then show we can compute the Schindler Transform [26] (ST) in the StreamSort model and the BWT in the Read-Write model, so the bound is achievable in them.

We start by reviewing some preliminary material in Section 2. In Section 3 we show how, for any constants  $c$  and  $\epsilon$  with  $1 \geq c \geq 0$  and  $\epsilon > 0$ , we can store  $s$  in  $nH_k(s) + O(\sigma^k n^{1-c+\epsilon})$  bits in the Standard model with  $O(n^c)$  bits of memory; we then show this bound is nearly optimal in the sense that we cannot always store  $s$  in, nor recover it from,  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon})$  bits. In Section 4 we extend our tradeoff to the Multipass and W-Streams models. Since we can store BWT( $s$ ) in, and recover it from,  $3.4nH_k(s) + O(\sigma^k)$  bits in even the Standard model, our tradeoff implies we can neither compute nor invert the BWT in the first three models. In Section 5 we show how we can compute the ST in the StreamSort model and in Section 6 we show how we can compute the BWT in the Read-Write model. Using either transform, we can store  $s$  in  $1.8nH_k(s) + O(\sigma^k \log n)$  bits in these models.

## 2 Preliminaries

Throughout this paper, we assume  $s = s_1 \cdots s_n$  is a string of length  $n$  over an alphabet of constant size  $\sigma$ , and  $c$  and  $\epsilon$  are constants with  $1 - \epsilon > c > \epsilon > 0$ . The 0th-order empirical entropy  $H_0(s)$  of  $s$  is the entropy of the characters' distribution in  $s$ , i.e.,  $H_0(s) = \frac{1}{n} \sum_{a \in s} n_a \log \frac{n}{n_a}$  where  $a \in s$  means character  $a$  occurs in  $s$  and  $n_a$  is its frequency. The  $k$ th-order empirical entropy  $H_k(s)$  of  $s$  for  $k \geq 1$ , described in detail by Manzini [21], is defined as

$$H_k(s) = \frac{1}{n} \sum_{|w|=k} |w_s| H_0(w_s),$$

where  $w_s$  is the concatenation of characters that immediately follow occurrences of  $w$  in  $s$ .

The BWT is an invertible transform that permutes the characters in  $s$  so that  $s_i$  comes before  $s_j$  if  $s_{i-1} \cdots s_{i-n}$  is lexicographically less than  $s_{j-1} \cdots s_{j-n}$ , taking indices modulo  $n+1$  and considering  $s_0 = s_{n+1}$  to be lexicographically less than any character in the alphabet. In other words, the BWT sorts  $s$ 's

characters in order of their contexts, which start at their predecessors and extend backwards. Although the distribution of characters remains the same — so  $H_0(\text{BWT}(s)) = H_0(s)$  — characters with similar contexts are moved close together so, if  $s$  is compressible with an algorithm that takes uses contexts, then  $\text{BWT}(s)$  is compressible with an algorithm that takes advantage of local homogeneity. Move-to-front [3] and distance coding [4,8] are two reversible transforms that turn strings with local homogeneity into strings of numbers with low 0th-order empirical entropy: move-to-front keeps a list of the characters in the alphabet and replaces each character in the input by its position in the list and then moves it to the front of the list; distance coding writes the distance to the first occurrence of each character and the length of the string, then replaces each character in the input by the distance from the last occurrence of that character, omitting 1s. (The numbers are often written with, e.g., Elias’ delta code [9], in which case move-to-front and distance coding become compression algorithms themselves.) Building on work by Kaplan, Landau and Verbin [16], we showed in another previous paper [11] that composing the BWT, move-to-front, run-length coding and arithmetic coding produces an encoding that contains at most  $3.4nH_k(s) + O(\sigma^k)$  bits for all  $k$  simultaneously; with the BWT, distance coding and arithmetic coding, the bound is  $1.8nH_k(s) + O(\sigma^k \log n)$  bits. Using the BWT in more sophisticated ways, Ferragina, Manzini, Mäkinen and Navarro [10] and Mäkinen and Navarro [20] achieved a bound on the encoding’s length of  $nH_k(s) + o(n/\log n)$  bits for all  $k$  at most a constant proper fraction of  $\log_\sigma n$ ; Grossi, Gupta and Vitter [14] achieved a bound of  $nH_k(s) + O(\sigma^k \log n)$  bits for all  $k$  simultaneously, matching a lower bound due to Rissanen [24]. In Section 3 we show how, because length times empirical entropy is superadditive — i.e.,  $|a|H_k(a) + |b|H_k(b) \leq |ab|H_k(ab)$  — we can trade-off between memory and redundancy by breaking  $s$  into blocks and encoding each block in turn with Grossi, Gupta and Vitter’s algorithm: the memory needed decreases by a factor roughly equal to the number of blocks, while the redundancy increases by that much. In Section 5 we use the ST instead of the BWT. When using contexts of length  $k$ , the ST permutes the characters of  $s$  so that  $s_i$  comes before  $s_j$  if  $s_{i-1} \cdots s_{i-k}$  is lexicographically less than  $s_{j-1} \cdots s_{j-k}$  or, when they are equal, if  $i < j$ . By the same arguments as for the BWT, composing the ST, distance coding and arithmetic coding produces an encoding that contains at most  $1.8nH_k(s) + O(\sigma^k \log n)$  bits for the chosen context length  $k$ .

A  $\sigma$ -ary De Bruijn sequence of order  $k \geq 1$  contains each possible  $k$ -tuple exactly once and, it follows, has length  $\sigma^k + k - 1$  and starts and ends with the same  $k - 1$  characters. Notice that, if  $d$  consists of the first  $\sigma^k$  characters of such a sequence, then  $H_k(d^i) = 0$  for any  $i$ . However, there are  $(\sigma!)^{\sigma^{k-1}}$  such sequences [6,17,18] so, by Li and Vitányi’s Incompressibility Lemma [19], for any fixed algorithm  $A$ , the maximum Kolmogorov complexity of  $d$  relative to  $A$  is at least  $\lfloor \sigma^{k-1} \log \sigma! \rfloor = \Omega(\sigma^k)$  bits. We used this fact in [12] to prove that a one-pass algorithm cannot always compress well in terms of the  $k$ th-order empirical without using memory exponential in  $k$ : we can compute  $d$  from the configurations of  $A$ ’s memory when it starts and finishes reading any copy of  $d$

in  $d^i$  and its output while reading that copy (if there were another string  $d'$  that took  $A$  between those configurations while producing that output, then we could substitute  $d'$  for that copy of  $d$  without changing the total encoding); therefore, if  $A$  uses  $o(\sigma^k)$  bits of memory, then its total output must be  $\Omega(|d^i|)$  bits.

### 3 The Standard model

We can easily store  $s$  in  $nH_0(s) + 2n + O(1)$  bits in one pass with  $O(\log n)$  bits of memory with dynamic Huffman coding [27], for example; by running a separate copy of the algorithm for each possible  $k$ -tuple for any given  $k$ , we can store  $s$  in  $nH_k(s) + 2n + O(\sigma^k)$  bits in one pass with  $O(\sigma^k \log n)$  bits of memory. With adaptive arithmetic coding instead of dynamic Huffman coding, the  $2n$  term becomes approximately  $n/100$  (see, e.g., [15]). With the version of LZ77 we described in [12], for any fixed  $k$  we can store  $s$  in  $nH_k(s) + o(n)$  bits in one pass with  $o(n)$  bits of memory, but with both  $o(n)$  terms nearly linear. We now show we can substantially reduce those terms simultaneously, answering a question we posed in that paper.

**Theorem 3.1.** *In the Standard model with  $O(n^c)$  bits of memory, we can store  $s$  in, and later recover it from,  $nH_k(s) + O(\sigma^k n^{1-c+\epsilon})$  bits for all  $k$  simultaneously.*

*Proof.* Let  $A$  be Grossi, Gupta and Vitter's algorithm. They proved  $A$  stores  $s$  in  $nH_k(s) + O(\sigma^k \log n)$  bits for all  $k$  simultaneously using  $O(n)$  time so, although they did not give an explicit bound on the memory used, we know it is at most  $A$ 's time complexity multiplied by the word size, i.e.,  $O(n \log n)$  bits.

First, suppose we know  $n$  in advance. We process  $s$  in  $O(n^{1-c+\epsilon/2})$  blocks  $b_1, \dots, b_m$ , each of length  $O(n^{c-\epsilon/2})$ : we read each block  $b_i$  in turn, compute and output  $A(b_i)$  — using  $O(|b_i| \log |b_i|) = O(n^c)$  bits of memory — and erase  $b_i$  from memory. As we noted in Section 2, empirical entropy is superadditive, so the total length of the encodings we output is at most

$$\sum_{i=1}^m (|b_i| H_k(b_i) + O(\sigma^k \log |b_i|)) \leq nH_k(s) + O(\sigma^k n^{1-c+\epsilon})$$

bits for all  $k$  simultaneously.

Now, suppose we do not know  $n$  in advance. We work as before but we start with a constant estimate of  $n$  and, each time we have read that many characters of  $s$ , we double it. This way, we increase the size of the largest block by less than 2 and the number of blocks by an  $O(\log n)$ -factor, so our asymptotic bounds on the memory used and the whole encoding's length does not change.  $\square$

Extending our lower bounds from that paper, we now show we cannot reduce the factor  $n^{1-c+\epsilon}$  much further unless we increase the factor  $\sigma^k$ , not even if we multiply the bound on the encoding's length by any constant coefficient. This lower bound holds for both compression and decompression; as we noted in [12], “good bounds [for decompression] are equally important, because often data

is compressed once by a powerful machine (e.g., a server or base-station) and then transmitted to many weaker machines (clients or agents) who decompress it individually.”

**Theorem 3.2.** *In the Standard model with  $O(n^c)$  bits of memory, we cannot store  $s$  in  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon})$  bits in the worst case for, e.g.,  $k = \lceil (c + \epsilon/2) \log_\sigma n \rceil$ .*

*Proof.* Consider any compression algorithm  $A$  that works in the Standard model with  $O(n^c)$  bits of memory. Suppose  $s$  consists of copies of the first  $\sigma^k$  characters  $d$  in a  $\sigma$ -ary De Bruijn sequence of order  $k = \lceil (c + \epsilon/2) \log_\sigma n \rceil$  whose Kolmogorov complexity relative to  $A$  is  $\Omega(\sigma^k) = \Omega(n^{c+\epsilon/2})$  bits. As we noted in Section 2, we can compute  $d$  from the configurations of  $A$ ’s memory when it starts and finishes reading any copy of  $d$  and its output while reading that copy. Since  $A$ ’s memory is asymptotically smaller than  $d$ ’s Kolmogorov complexity relative to  $A$ , it must output  $\Omega(\sigma^k)$  bits for every copy of  $d$  in  $s$ , i.e.,  $\Omega(n)$  bits altogether. Since  $H_k(s) = 0$ , however,  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon}) = O(n^{1-\epsilon/2})$ .  $\square$

**Theorem 3.3.** *In the Standard model with  $O(n^c)$  bits of memory, we cannot recover  $s$  from  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon})$  bits in the worst case for, e.g.,  $k = \lceil (c + \epsilon/2) \log_\sigma n \rceil$ .*

*Proof.* Let  $A$  be a decompression algorithm that works in the Standard model with  $O(n^c)$  bits of memory and let  $s$  be as in the proof of Theorem 3.2. We can also compute  $d$  from the configuration of  $A$ ’s memory when it starts outputting any copy of  $d$  and the bits it reads while outputting that copy; since  $A$ ’s memory is asymptotically smaller than  $d$ ’s Kolmogorov complexity relative to  $A$ , it must read  $\Omega(\sigma^k)$  bits for each copy of  $d$  in  $s$ , i.e.,  $\Omega(n)$  bits altogether, whereas  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon}) = O(n^{1-\epsilon/2})$ .  $\square$

Finally, we now prove that, if we could compute the BWT in the Standard model, then we could achieve the bounds we have just proven unachievable; we will draw the obvious conclusion in Section 4 as part of a more general theorem.

**Lemma 3.4.** *In the Standard model with  $O(\log n)$  bits of memory, we can store  $\text{BWT}(s)$  in, and later recover it from,  $3.4nH_k(s) + O(\sigma^k)$  bits for all  $k$  simultaneously.*

*Proof.* We encode  $\text{BWT}(s)$  by composing move-to-front, run-length coding and adaptive arithmetic coding; since encoding or decoding each of the three steps takes one pass and  $O(\log n)$  bits of memory, so does their composition. As we noted in Section 2, the resulting encoding contains at most  $3.4nH_k(s) + O(\sigma^k)$  bits for all  $k$  simultaneously.  $\square$

## 4 The Multipass and W-Streams models

We now extend our tradeoff to the Multipass and W-Streams models. Of course, anything we can do in the Standard model we can do in those models, so the upper bounds are immediate.

**Theorem 4.1.** *In the Multipass and W-Streams models with  $O(n^c)$  bits of memory and one pass, we can store  $s$  in  $nH_k(s) + O(\sigma^k n^{1-c+\epsilon})$  bits for all  $k$  simultaneously.*

Conversely, lower bounds for the W-Streams model apply to the Multipass model. We could quite easily extend our proofs to include the Multipass model alone: e.g., to see we cannot compress  $s$  very well with polylogarithmic passes, notice we can compute  $d$  from the configurations of  $A$ 's memory when it starts and finishes reading any copy of  $d$  and its output while reading that copy *during each pass*; since  $\log^{O(1)} n = o(n^{\epsilon/2})$ , even a polylogarithmic number of  $A$ 's memory configurations are asymptotically smaller than  $d$ 's Kolmogorov complexity relative to  $A$ , so the rest of our argument still holds. The proof for the W-Streams model must be slightly different because, for example, after the first pass the tape will generally not contain copies of  $d$ .

**Theorem 4.2.** *In the Multipass and W-Streams models with  $O(n^c)$  bits of memory and  $\log^{O(1)} n$  passes, we cannot store  $s$  in  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon})$  bits in the worst case for, e.g.,  $k = \lceil (c + \epsilon/2) \log_\sigma n \rceil$ .*

*Proof.* We need consider only the more general W-Streams model. Consider any compression algorithm  $A$  that works in the W-Streams model with  $O(n^c)$  bits of memory and  $\log^{O(1)} n$  passes, and let  $s$  be as in the proof of Theorem 3.2. Without loss of generality, assume  $A$ 's output consists of the contents of the tape after its last pass (any output from intermediate passes can be written on the tape instead). Notice any substring of characters on the tape immediately after a particular pass must have been written (or left untouched) while  $A$  was reading a substring of characters on the tape immediately before that pass. Suppose  $A$  makes  $p$  passes over  $s$ . Consider any copy  $d_0$  of  $d$  in  $s$  and, for  $p \geq i \geq 1$ , let  $d_i$  be the substring  $A$  writes while reading  $d_{i-1}$ . We claim we can compute  $d$  from  $d_p$  and the memory configurations of  $A$  when it starts and finishes reading each  $d_i$ . To see why, suppose there were a sequence  $d'_0 \neq d_0, d'_1, \dots, d'_{p-1}, d'_p = d_p$  such that, for  $p \geq i \geq 1$ ,  $d'_{i-1}$  took  $A$  between the  $i$ th pair of memory configurations while writing  $d'_i$ ; then we could substitute  $d'_0$  for  $d_0$  without changing the total encoding. It follows that  $d_p$  must contain  $\Omega(\sigma^k)$  bits and, so, the whole tape must contain  $\Omega(n)$  bits after the last pass, whereas  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon}) = O(n^{1-\epsilon/2})$ .  $\square$

**Theorem 4.3.** *In the Multipass and W-Streams models with  $O(n^c)$  bits of memory and  $\log^{O(1)} n$  passes, we cannot recover  $s$  from  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon})$  bits in the worst case for, e.g.,  $k = \lceil (c + \epsilon/2) \log_\sigma n \rceil$ .*

*Proof.* Again, we consider only the W-Streams model. Consider any decompression algorithm  $A$  that works in the W-Streams model with  $O(n^c)$  bits of memory and  $\log^{O(1)} n$  passes, and let  $s$  be as in the proof of Theorem 3.2. Without loss of generality, assume the tape contains  $s$  after  $A$ 's last pass. Consider any copy  $d_0$  of  $d$  in  $s$  and, for  $p \geq i \geq 1$ , let  $d_i$  be the substring  $A$  reads while writing  $d_{i-1}$ ; notice this is the reverse of the definition in the proof of Theorem 4.2.

Notice we can compute  $d$  from  $d_p$  and the memory configuration of  $A$  when it starts reading each  $d_i$ : running  $A$  on  $d_i$ , starting in the memory configuration, produces  $d_{i-1}$ . It follows that  $A$  must read  $\Omega(\sigma^k)$  bits for each copy of  $d$  in  $s$ , i.e.,  $\Omega(n)$  bits altogether, whereas  $O(nH_k(s) + \sigma^k n^{1-c-\epsilon}) = O(n^{1-\epsilon/2})$ .  $\square$

We note in passing that in the W-Streams model, we can easily reduce sorting the characters in  $s$  to computing the BWT: we compute  $s' = (s_1, s_0)(s_2, s_1) \cdots (s_n, s_{n-1})(s_0, s_n)$ ; we compute  $\text{BWT}(s') = (s_1, s_0)(s_{i_1+1}, s_{i_1}) \cdots (s_{i_n+1}, s_{i_n})$ ; and we output  $s_{i_1}, \dots, s_{i_n}$ . To see why  $s_{i_1}, \dots, s_{i_n}$  are sorted, suppose  $(s_{i+1}, s_i)$  precedes  $(s_{j+1}, s_j)$  in  $\text{BWT}(s')$ . The BWT arranges the pairs in  $s'$  in the lexicographic order of their predecessors (how it breaks ties does not concern us now), which is that of their predecessors' first components, or that of their own second components — so  $s_i \leq s_j$ . If  $\sigma$  were unbounded, this reduction would imply we could not compute the BWT in the first three models; since  $\sigma$  is constant, however, it is meaningless — we can sort the characters in  $s$  in the Multipass model anyway. Fortunately, we can also easily reduce storing  $s$  in  $O(nH_k(s) + \sigma^k \log n)$  bits to computing the BWT.

**Theorem 4.4.** *In the Standard, Multipass and W-Streams models with  $\log^{O(1)} n$  bits of memory and passes, we can neither compute nor invert  $\text{BWT}(s)$  in the worst case.*

*Proof.* If we could compute or invert  $\text{BWT}(s)$  then, by Lemma 3.4, we could store  $s$  in, or recover it from,  $3.4nH_k(s) + O(\sigma^k)$  bits for all  $k$  simultaneously; however, by Theorems 3.2, 3.3, 4.2 and 4.3 we cannot achieve this bound in the worst case.  $\square$

## 5 The StreamSort model

The ST is known as both the “Schindler Transform” and the “Sort Transform”, so it is perhaps not surprising that it can be computed in the StreamSort model. Indeed, computing the ST for any given  $k = O(\log n)$  takes only a constant number of passes once we have padded the input from  $O(n)$  bits to  $\Omega(n \log n)$  bits; we do this padding, which takes  $O(\log \log n)$  passes, because we are allowed to expand the tape contents by a only constant factor during each pass, and we want to eventually associate each character with a  $O(\log n)$ -bit key — the  $k$ -tuple that precedes that character in  $s$  — and then stably sort by the keys. Unfortunately, we do not know yet how to invert the ST in either the StreamSort or Read-Write models.

**Lemma 5.1.** *For any given  $k = O(\log n)$ , we can compute  $\text{ST}(s)$  in the StreamSort model with  $O(\log n)$  bits of memory and  $O(\log \log n)$  passes.*

*Proof.* We make  $O(\log \log n)$  passes, each time doubling the length of each character's representation by padding it with 0s, until each character takes  $\Omega(\log n)$  bits. We make another pass to associate each character in  $s$  with the  $k$ -tuple that precedes it; since  $\log \sigma^k = O(\log n)$ , we can use  $O(\log n)$  bits of memory to

keep track of the last  $k$  characters we have seen and write them as a key in front of the next character while only doubling the number of bits on the tape. We then use  $O(1)$  passes to stably sort by those keys — i.e., computing the ST — and, finally, delete the keys and padding.  $\square$

**Theorem 5.2.** *In the StreamSort model with  $O(\log n)$  bits of memory and  $O(\log n \log \log n)$  passes, we can store  $s$  in  $1.8nH_k(s) + O(\sigma^k \log n)$  bits for all  $k$  simultaneously.*

*Proof.* We compute  $ST(s)$  in  $O(\log \log n)$  passes and encode it in  $O(1)$  passes by composing distance coding and adaptive arithmetic coding, for each  $k = O(\log n)$ ; in total, we use  $O(\log n)$  bits of memory and  $O(\log n \log \log n)$  passes. As we noted in Section 2, each resulting encoding contains at most  $1.8nH_k(s) + O(\sigma^k \log n)$  bits for the value of  $k$  used to compute it; thus, the shortest encoding contains  $1.8nH_k(s) + O(\sigma^k \log n)$  for all  $k = O(\log n)$  simultaneously and so — because

$$1.8nH_0(s) + O(\log n) < 1.8nH_k(s) + (\sigma^k \log n) = \omega(n)$$

for all  $k = \omega(\log n)$  — for all  $k$  simultaneously.  $\square$

## 6 The Read-Write model

Anything we can do in the StreamSort model we can do in the Read-Write model using an  $O(\log n)$ -factor more passes, so Theorem 5.2 implies we can store  $s$  in  $1.8nH_k(s) + (\sigma^k \log n)$  bits for all  $k$  simultaneously in the Read-Write model with  $O(\log n)$  bits of memory and  $O(\log^2 n \log \log n)$  passes. It does not, however, imply we can recover  $s$  again in the Read-Write model. Fortunately, using techniques based on the doubling algorithm by Arge, Ferragina, Grossi and Vitter [2] for sorting strings in external memory (see also [7]), we can both compute and invert  $BWT(s)$  in the Read-Write model. Figures 1 and 2 show how we compute and invert  $BWT(\text{mississippi})$ ; to save space, Figure 2 shows two rounds of the algorithm in each row.

**Theorem 6.1.** *In the Read-Write model with  $O(\log n)$  bits of memory and  $O(\log^2 n)$  passes, we can store  $s$  in, and later recover it from,  $1.8nH_k(s) + O(\sigma^k \log n)$  bits for all  $k$  simultaneously.*

*Proof.* As we noted in Section 2, once we have  $BWT(s)$ , we can store it in  $1.8nH_k(s) + O(\sigma^k \log n)$  bits for all  $k$  simultaneously by composing distance coding and adaptive arithmetic coding. To compute or invert distance coding and to encode or decode adaptive arithmetic coding all take  $O(\log n)$  bits of memory and  $O(1)$  passes. Therefore, we need consider only how to compute and invert  $BWT(s)$ . Due to space constraints, here we only sketch these procedures; we will give full descriptions and analyses in the full paper.

To compute  $BWT(s)$ , we append a special character that is lexicographically less than any character in the alphabet ( $\#$  in Figures 1 and 2). We tag each character in  $s$  with a unique identifier (e.g., its position; in this model, we can



**Fig. 1.** Computing the BWT in the Read-Write model.

**Fig. 2.** Inverting the BWT in the Read-Write model.

expand the contents of the tape by more than a constant factor during a pass, so we do not need to pad — although  $O(\log \log n)$  extra rounds would not make a difference here anyway), then form a triple from it by appending a 1 and its successor in  $s$ . We make two copies of the set of triples, sort the first copy by the last component and the second copy by the first component (breaking ties by characters identifiers), and merge them to form quintuples. (It is the copying and sorting step that we do not see how to do in the StreamSort model.) We sort the set of quintuples by the fourth component, breaking ties by the third component (ignoring characters' identifiers), breaking continued ties by the second component, and breaking continued ties arbitrarily. (In the first round, all the fourth and second components are 1, so we effectively sort by the third component; notice the third component is the first component's successor in  $s$  and the fifth component's predecessor in  $s$ , taking indices modulo  $n + 1$ .) Finally, we replace the middle triples — the second, third and fourth components — with numbers, starting at one and incrementing whenever we find a triple different from the one before. This process results in another set of triples; if the second components are the numbers 1 through  $n + 1$ , we stop; otherwise, we repeat the procedure from the point of copying the triples.

Notice that, at the end of the first round, the first and third components in any triple are two positions apart in  $s$ , taking indices modulo  $n + 1$ . Also, for any two triples  $(s_i, x, s_{i+2})$  and  $(s_j, y, s_{j+2})$ , the comparative relationship between  $x$  and  $y$  is the same as the lexicographic relationship between  $s_{i+1}$  and  $s_{j+1}$ . At the end of the second round, the first and third components in any triple are four positions apart in  $s$  and, for any two triples  $(s_i, x, s_{i+4})$  and  $(s_j, y, s_{j+4})$ , the comparative relationship between  $x$  and  $y$  is the same as the lexicographic relationship between  $s_{i+3}s_{i+2}s_{i+1}$  and  $s_{j+3}s_{j+2}s_{j+1}$ . To see why, notice the relationship between  $x$  and  $y$  depends, in decreasing order of priority, on the relationships between  $x_1$  and  $y_1$ ,  $s_{i+2}$  and  $s_{j+2}$ , and  $x_2$  and  $y_2$  in the quintuples  $(s_i, x_2, s_{i+2}, x_1, s_{i+4})$  and  $(s_j, y_2, s_{j+2}, y_1, s_{j+4})$ ; since these quintuples are formed by joining triples  $(s_i, x_2, s_{i+2})$  and  $(s_{i+2}, x_1, s_{i+4})$  and  $(s_j, y_2, s_{j+2})$  created during the first round, the comparative relationships between  $x_1$  and  $y_1$  and  $x_2$  and  $y_2$  are the same as the lexicographic relationships between  $s_{i+3}$  and  $s_{j+3}$  and  $s_{i+1}$  and  $s_{j+1}$ . After  $O(\log n)$  rounds, when the second components are the numbers 1 through  $n$ , they indicate the lexicographic relationships of the prefixes of the third components, so the third components are  $\text{BWT}(s)$ — in our example in Figure 1,  $\text{BWT}(s) = \text{ms\#spipissii}$ . (We note that, if we sorted quintuples by the second component, using the third and fourth to break ties, then we would compute the suffix array of  $s$ .) We need only  $O(\log n)$  bits of memory for this procedure; each sorting step takes  $O(\log n)$  passes and each other step takes  $O(1)$  passes, so we use  $O(\log^2 n)$  passes altogether.

To invert  $\text{BWT}(s)$ , we again tag each character in  $\text{BWT}(s)$  with a unique identifier and form triples. This time, however, we prepend a ? to each character, then prepend the corresponding character in the stable sort of  $\text{BWT}(s)$ ; finally, in the triple whose first component is the special character not in the alphabet, we replace the ? by  $n + 1$ . As for computing  $\text{BWT}(s)$ , we make two copies of

the set of triples, sort them and merge them. This time, for each triple, if the second component is not a ? or the third component is a ?, then we simply delete the third and fourth components; if the second component is a ? and the third component is, then we put the third component minus 1 in the second component, then delete the third and fourth components. This process results in another set of triples; if the second components are the numbers 1 through  $n + 1$  in some order, we stop; otherwise, we repeat the procedure from the point of copying the triples.

By the definition of the BWT, the  $i$ th character in the stable sort of  $\text{BWT}(s)$  is the predecessor in  $s$  of the  $i$ th character in  $\text{BWT}(s)$ . Therefore, at the end of the first round, the first and third components in any triple are two positions apart in  $s$ , taking indices modulo  $n + 1$ ; also, the triples that have  $s_n$  and  $s_{n+1}$  as their first components have  $n$  and  $n + 1$  as their second components. At the end of the second round, the first and third components of any triple are 4 positions apart in  $s$  and the triples that have  $s_{n-2}$ ,  $s_{n-1}$ ,  $s_n$  and  $s_{n+1}$  as their first components have  $n - 2$ ,  $n - 1$ ,  $n$  and  $n + 1$  as their second components. After  $O(\log n)$  rounds, when the second components are the numbers from 1 through  $n + 1$  in some order, they indicate the positions in  $s$  of the triples' first components. Sorting by the second component and ignoring the special character, we can recover  $s$ . In our example in Figure 2, the triple that starts 'm' then has a 1 as its second component; the triples that start 'i' then have 2, 5, 8 and 11; the triples that start 'p' then have 9 and 10; and the triples that start 's' then have 3, 4, 6 and 7; thus, sorting them by the second component and outputting the first, we recover mississippi. Again, we need only  $O(\log n)$  bits of memory for this procedure; each sorting step takes  $O(\log n)$  passes and each other step takes  $O(1)$  passes, so we use  $O(\log^2 n)$  passes altogether.  $\square$

Grohe and Schweikardt showed that, given a sequence of  $n$  numbers  $x_1, \dots, x_n$ , each of  $2 \log n$  bits, we cannot generally sort them using  $o(\log n)$  passes,  $O(n^{1-\epsilon})$  bits of memory and  $O(1)$  tapes, for any positive constant  $\epsilon$ . We can easily obtain the same lower bound for the BWT, via the following reduction from sorting: given  $x_1, \dots, x_n$ , using one pass,  $O(\log n)$  bits of memory and two tapes, for  $1 \leq i \leq n$  and  $1 \leq j \leq 2 \log n$ , we replace the  $j$ th bit  $x_i[j]$  of  $x_i$  by  $x_i[j] \ 2 \ x_i \ i \ j$ , writing 2 as a single character,  $x_i$  in  $2 \log n$  bits,  $i$  in  $\log n$  bits and  $j$  in  $\log \log n + 1$  bits; notice the resulting string is of length  $n(3 \log n + \log \log n + 2)$ . (For simplicity, we now consider each character's context as starting at its successor and extend forwards.) The only characters followed by 2s in this string are the bits at the beginning of replacement phrases so, if we perform the BWT of it, the last  $2n \log n$  characters of the transformed string are the bits of  $x_1, \dots, x_n$ ; moreover, since the lexicographic order of equal-length binary strings is the same as their numeric order, the  $x_i[j]$ s will be arranged by  $x_i$ s, with ties broken by the  $i$ s (so if  $x_i = x_{i'}$  with  $i < i'$ , then every  $x_i[j]$  comes before every  $x_{i'}[j']$ ), and further ties broken by the  $j$ s; thus, the last  $2n \log n$  bits of the transformed string are  $x_1, \dots, x_n$  in sorted order.

## References

1. G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science*, pages 540–549, 2004.
2. L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 540–548, 1997.
3. J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.
4. E. Binder. Distance coding. Usenet group comp.compression, April 13th, 2000.
5. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 24, Digital Equipment Corporation, 1994.
6. N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie van Wetenschappen*, 49:758–764, 1946.
7. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, to appear.
8. S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software - Practice and Experience*, 32:99–111, 2002.
9. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.
10. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3, 2007.
11. T. Gagie and G. Manzini. Move-to-front, distance coding, and inversion frequencies revisited. In *Proceedings of the 18th Symposium on Combinatorial Pattern Matching*, pages 71–82, 2007.
12. T. Gagie and G. Manzini. Space-conscious compression. In *Proceedings of the 32nd Symposium on Mathematical Foundations of Computer Science*, pages 206–217, 2007.
13. M. Grohe and N. Schweikardt. Lower bounds for sorting with few random accesses to external memory. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems*, pages 238–249, 2005.
14. R. Grossi, A. Gupta, and J. S. Vitter. An algorithmic framework for compression and text indexing. Personal communication.
15. P. Howard and J. S. Vitter. Analysis of arithmetic coding for data compression. *Information Processing and Management*, 28:749–763, 1992.
16. H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler based compression. *Theoretical Computer Science*, 387:220–235, 2007.
17. D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 3rd edition, 1997.
18. D. E. Knuth. *The Art of Computer Programming*, volume 4, fascicle 2. Addison-Wesley, 2005.
19. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2nd edition, 1997.
20. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, to appear.
21. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48:407–430, 2001.

22. J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.
23. S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers, 2005.
24. J. Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, 29:656–663, 1983.
25. M. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, 2003.
26. M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the 1997 IEEE Data Compression Conference*, page 469, 1997.
27. J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34:825–845, 1987.
28. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.